Multi-Layered Adaptation for the Failure Prevention and Recovery in Cloud Service Brokerage Platforms

Nicolas Ferry, Franck Chauvel, Brice Morin

Citation:
Nicolas Ferry, Franck Chauvel, Brice Morin(2018). Multi-Layered Adaptation for the Failure Prevention and Recovery in Cloud Service Brokerage Platforms. In 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), Coimbra, Portugal, 4-7 Sept. 2018, pp 21-29 DOI: 10.1109/QUATIC.2018.00014

Multi-Layered Adaptation for the Failure Prevention and Recovery in Cloud Service Brokerage Platforms

Nicolas Ferry, Franck Chauvel, Brice Morin
SINTEF, Oslo, Norway. Email: first.last@sintef.no

***Abstract*—Self-adaptation is a basic capability of modern applications, which adjust their structure and behaviour at run-time, adapting to changes in their environment, in order to maintain the quality of service at runtime. Models@run-time is an emerging approach for adaptation, whereby a models@run-time engine maintains a causal connection between an application model and the running application, so that a reasoner can adapt the application structure and behaviour by reading and writing this model. However, when used on the dynamic quality control of cloud-based applications, a traditional first-order adaptation is usually not sufficient. This is because during the application life cycle, its requirements may also change, which requires adaptation on the reasoner itself. In this paper, we propose a multi-layered models@run-time approach to enable a second-order adaptation. By maintaining a causal connection between an adaptation model, which reflects the behaviour of the reasoner, and the running application, we enable the adaptation to be automatically adjusted according to the changes in the running application. We apply this approach on a case study for failure prevention and recovery in cloud service brokerage platforms.**

***Keywords*-models@run-time; multi-layer; failure prevention; failure recovery; brokerage; cloud computing;**

I. Introduction

Dynamic adaptability is an important feature for modern software applications to maintain an expected quality at runtime [1], and model-driven engineering is playing an increasing role in realizing it [2]. In particular, models@run-time is one of the key techniques to support model-based dynamic adaptation [3]. A change in the running system is automatically reflected in the run-time model of the target running system. Similarly, a modification to this model is enacted on the running system on demand. Thanks to this causal connection, an adaptation reasoner can monitor and adapt the target system in the same way as reading and writing the model. Automatic adaptation reasoners often exploit another model to guide their own analysis and planning behaviour. In this paper we refer to such a model as an *adaptation model* in order to distinguish it from the traditional *run-time model*. Adaptation models are typically at a higher abstraction level than the run-time model, derived from the earlier software development phases, such as state machines [4], aspect models [5], variability models [6], [7], etc. Adaptation models determines the behaviour of the adaptation reasoners, and the reasoners adapt the running systems by manipulating the lower-level run-time models representing architectures, configurations, events, etc.

In traditional models@run-time, the adaptation model of the reasoner is not causally connected to the running system. Hence, the adaptation behaviour (*i.e.*, how to analyse and plan the running system) is specified at design-time and cannot be changed automatically at run-time. This is because the adaptation behaviour is assumed not to be affected by the running system. However, this assumption does not necessarily apply to all systems [8]. Some changes on these systems may require the behaviour of the adaptation reasoner to be adjusted as well. This calls for a second order adaptation.

Many systems in the cloud computing paradigm require such second order adaptation. Typical examples are the public supportive services for cloud system operations, such as the cloud resource portals, the monitoring consoles, the service brokerage platforms, etc. These systems have a long life-cycle, and are not allowed to be stopped during run-time. More importantly, during the long life-cycle, the content managed by these systems and the service level agreements with the users are all subject to change. In the paper, we use a typical cloud supportive service, the cloud service brokerage, as an example, and focus on a particular type of adaptation for the purpose of quality assurance at runtime, *i.e.*, the failure prevention and recovery of services hosted on the platform. At the first layer, adaptations consist in recommending substitution of services when some of them are about to fail or have failed. This recommendation is based on the variability existing in the broker, in terms of alternative services, the relation between them, and their properties. As an open platform, the broker allows service providers to register new alternative services, reset their relations, or introduce new properties to describe the system. When such changes happen, the variability in the system evolves. As a result, the adaptation reasoner has to evolve as well, so that the new recommendations consider the newly introduced services, relations or properties.

In this paper, we present our approach towards *multi-layered* models@run-time in order to support higher-order adaptation in a model-based way: Both the traditional run-time model and the adaptation model are causally connected to the target system. This enables an automatic second order adaptation, where the behaviour of the adaptation reasoner is automatically adjusted at run-time based on the changes in the running system. The main challenges for such multi-layered models@run-time in general, are the lack of knowledge to maintain the adaptation model, and the large abstraction gap between the adaptation model and the running system.

The contributions of this paper can be summarized as follows.

- We extend our earlier DiVA approach [6] to achieve two-

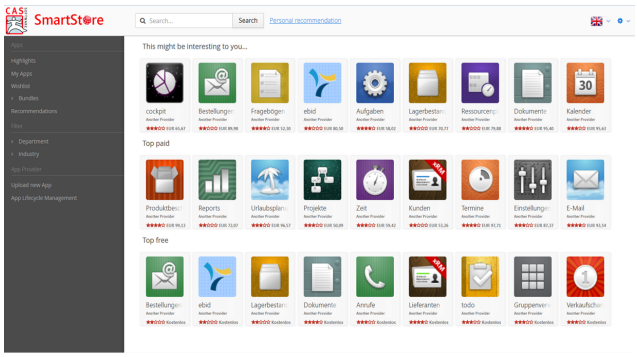

Figure 1. A snapshot of a sample service broker

layered variability model at run-time, and apply it to realise failure prevention and recovery in cloud service brokerage platforms.

- We implement a metamodel-driven approach to specify and maintain the causal connection between the service brokerage platform and the DiVA variability model.

In the remainder of the paper, we first give a brief introduction to the case study in Section II. Based on the requirement of the case study, Section III shows (i) how we leverage our former DiVA [6] solution, and (ii) why we need to extend it to support a second-order adaptation. Section IV presents how we implement this extension by supporting the causal connection between the DiVA model and the running system. Finally, Section VI discusses related approaches and Section VII concludes the paper.

II. Failure Prevention and Recovery of Service Brokerage Platforms

As cloud computing becomes more and more popular, there is a rapidly increasing set of cloud services that can be easily instantiated and used by consumers, and many of them have similar features and are able to be alternative to each other in some contexts. The problem becomes how to find the proper services for a special requirement. A service broker is a platform that aggregates such cloud services, together with their specifications. Service consumers can search, compare and choose the proper services, assisted by the searching, displaying, recommendation functions provided by the broker. At the same time, the broker also provides functions for service providers to register and advertise their services.

Figure 1 shows a snapshot of the GUI of a broker developed within the Broker@Cloud project. Each icon represents a service advertised in the broker. Some services are shown with the same icon, which means that they provide the same functional features, but with different quality of services (QoS). Such alternative services can be either different versions of the same service or different services developed by different providers.

Within the service broker, the variability mainly comes from the possibility of selecting alternatives services. For a particular required feature, the consumer may have a set of variants to choose from. In addition, specific functional requirements typically demand multiple features, implying the need to compose multiple services in order to fulfil the requirements. In the cases of service compositions, the variability can significantly increase. In the service broker developed in the Broker@Cloud project we exploit such variability to realise a failure prevention and recovery mechanism. When a service is about to fail or has failed, the broker recommends the consumers to switch to an alternative service, aiming to satisfy the original functional requirements and to adhere to the non-functional requirements as much as possible.

Table I
List of services in the sample broker

name	description
`GoldenOrbi`	fast response, high price, requires `CluDB` and `EMail`
`SilverOrbi`	medium response, high price, requires `CluDB` or `MonoDB`
`BronzeOrbi`	slow response, low price, requires `MonoDB`
`CluDB`	large storage, high price
`MonoDB`	small storage, low price
`EMail`	medium price

For the sake of simplicity we consider a broker with a limited set of alternative services as depicted in Table I. The `Orbi` service is a lightweight personal information management application. The application is provided in three different versions, with different prices and performances. We consider response time as the sole example for performance. A `GoldenOrbi` requires `CluDB` (for Clustered Database) to store data, and depends on an `EMail` client to provide the function of personal email integration. A `SilverOrbi` can rely on either a `CluDB` or a `MonoDB` (for Monolithic Database). The difference between two database services is on their storage volume and price. For a consumer, the variability reflects the different compositions of services that he or she can choose. Suppose the consumer's requirement is personal information management, then his choice could be [`GoldenOrbi`, `CluDB`, `EMail`], [`SilverOrbi`, `CluDB`], [`SilverOrbi`, `MonoDB`], or [`BronzeOrbi`, `MonoDB`].

When the broker detects that one of the composed services is failing, it needs to recommend the consumer to switch to one of the alternative compositions, until the provider fixes the problem. For example, suppose the current service composition is [`SilverOrbi`, `CluDB`], and the broker detects that `CluDB` is failing, then an obvious recommendation is to switch to [`SilverOrbi`, `MonoDB`]. The exact recommendation is to switch CluDB to MonoDB. Similarly, if the current composition is [`GoldenOrbi`, `CluDB`, `EMail`], and again CluDB fails, then the solution can be either [`SilverOrbi`, `MonoDB`] or [`BronzeOrbi`, `MonoDB`].

The failure prevention mechanism aims at detecting services with a high likelihood to fail in specific contexts, and to propose an alternative composition before the failure actually occurs and affects the consumer. Following the first scenario as we discussed above, from the current composition of [`SilverOrbi`, `CluDB`], if the broker detects that `CluDB` has a high likelihood to fail, it may substitute `MonoDB` for `CluDB`, if the benefit overcomes the compromise on the quality of service. Moreover, failure prevention also takes into

account the reason of failures. For example, if the current composition is [BronzeOrbi, MonoDB], BronzeOrbi is detected to be about to fail, and the reason is related to storage overload, then the proper alternative is [SilverOrbi, CluDB] because it has larger storage volume.

III. Towards Adaptable Variability Models at Run-time

In this section we introduce a typical variability model for a service broker using our previous approach DiVA (Dynamic Variability in Complex, Adaptive Systems) and exploit our case study to discuss why the traditional approach are not sufficient for building such open-adaptive systems. Then we introduce an two-layered *variability model at run-time* approach, where the variability model itself is automatically updated according to system changes.

A. First-order adaptation: variability models **for** run-time

Variability modelling allows describing the multiple variants of a system, by capitalizing on commonalities and formalizing the differences, rather than by enumerating the whole list of possible variants. In software engineering, variability modelling is closely related to software product line techniques.

Our previous work, DiVA [6] is a typical approach defining variability models *for* run-time to support dynamic adaptation. In DiVA, the variability of a system is described as a DiVA model. It describes a system by a number of Dimensions, each of them being a variability point. Multiple Variants are defined under each Dimension. For each Variant, the model describes its relation with other variants or a predefined global context variables, and a set of property values. Based on such a model, the DiVA reasoner computes proper configurations based on the current context. Each context is a vector of values assigned to the defined context variables. A configuration is a subsets of all the defined variants. Under each context, the DiVA reasoner ranks the valid configurations according to the property values of the selected variants.

Table II shows a simplified DiVA model for the case study we described before. The first three parts are used as the reference for the DiVA reasoner, and the reasoning result is illustrated by the last part, the context-configuration model. In the first part, we define the context variables corresponding to the availability of services, the requirement from consumers, and the possible reason of failure. In the second part, we define each service as a variant. The six variants are grouped into three dimensions, and the ones under the same dimension represent alternatives to each other. For each variant, we describe three non-functional properties, *i.e.*, price, storage (for storage capacity), and resp (for response time). Some properties are not relevant to all variants, this is marked by "-". We also define for each variant its relations with other variants (the "dependency" column), and with the global context variables (the "availability" and "requirement" columns, for sufficient and necessary conditions, respectively). In the third part, we define a set of priority rules. Each rule sets priority values for all the properties, which determine how important each property is when ranking the variants. A minus "-" means that a smaller value is regarded as better. Each rule applies to a condition, defined by an expression on context variables.

In the last part we present three sample contexts and their corresponding valid configurations. For example, the first row means that if all the services are available, and the requirement is simply to provide a Orbi service, we can use GoldenOrbi together with CluDB and EMail, among other valid configurations. However, if CluDB is not available (which means it is not running properly), there are only two valid configurations left, as shown in the second row. Finally, if the consumer requires Orbi and EMail support at the same time, there is only one choice. The DiVA reasoner computes the configurations by converting the whole DiVA model into a constraint satisfaction problem, solved by the Alloy constraint solver [9]. The solver will usually produce a number of valid configurations, and the reasoner will rank them based on the property value of each variant together with the priority of each property.

The default way of using DiVA for run-time adaptation is as shown in Figure 2(a). The DiVA model (the first three parts of Table II) is specified in advance at design-time and is used as the adaptation model. Then, during run-time, the monitoring mechanism provided by the broker extracts the recorded customer requirement and detects failed services, and creates a context accordingly. The DiVA reasoner takes the new context as input, and outputs a new configuration. Finally, the recommendation mechanism of the broker compares the new configuration with the original one, and generates service substitution commands.

B. Second-order adaptation: variability models **at** run-time

By default, when using the DiVA solution, the variability of the system is defined at design-time before the system is running and the variability model is not subject to change during run-time. However, this is not satisfactory in case of the service broker. As an open platform, the broker allows service providers to register new services, resetting the relation between services, and even introducing new types of properties. These changes ought to impact the allowed variability of the system, and affect the service substitution recommendation for the same contexts. Such variability model changes *at* run-time can be summarized into four categories.

Changing property values. Considering the failure prevention scenario that we discussed in Section II, a service with higher likelihood to fail should be ranked lower. In a DiVA model, this implies to add a new column of *failure likelihood* in the variants definition model, and the value of this property for each variant should then be modified during run-time.

Modifying the set of alternatives. A service provider may register a new service to be considered by the broker at any time, as an alternative variant to some existing services. This will usually result in furthermore configurations. For example, if a provider introduces a new FreeEMail service, then there should be a new row under the EMail dimension, and for the last context, there will be two configurations instead of one.

Table II
DIVA MODEL FOR SERVICE BROKERAGE PLATFORM

context variables						
GoldenOrbiAvail, SilverOrbiAvail, ..., EmailAvail, ReqOrbi, LargeDB, ReqEMail, LowResponse, CpuOverload						
dimensions and variants						
name	price	storage	resp	dependency	availability	requirement
Orbi: dimension						
GoldenOrbi	high	-	high	CluDB and EMail	GoldenOrbiAvail	RequiredOrbi
SilverOrbi	high	-	medium	CluDB or MonoDB	SilverOrbiAvail	RequiredOrbi
BronzeOrbi	low	-	low	MonoDB	BronzeOrbiAvail	RequiredOrbi
DB: dimension						
CluDB	high	high	-	-	CluDBAvail	LargeDB
MonoDB	low	low	-	-	MonoDBAvail	-
EMailDim: dimension						
EMail	medium	-	-	-	EmailAvail	ReqEMail
priority rules						
name	price	storage	resp	condition		
normal	-medium	medium	medium	-		
low resp.	-medium	medium	high	LowResponse		

contexts and configurations	
context	**configurations**
*Avail, ReqOrbi	[GoldenOrbi, CluDB, Email], [GoldenOrbi, CluDB], [SilverOrbi, MonoDB]...
*Avail, ¬CluDB, ReqOrbi	[SilverOrbi, MonoDB], [BronzeOrbi, MonoDB]
*Avail, ReqOrbi, ReqEMail	[GoldenOrbi, CluDB, Email]

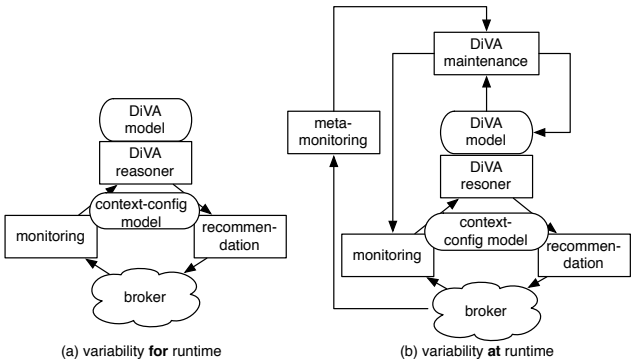

Figure 2. Adapting the Variability at run-time

New dimension to place a new variant (in this case a new service model). For example, a provider can register a Calendar service as a plugin to `GoldenOrbi`. This leads to a new variant and a new dimension. In addition, there should be a new requirement `CalendarReq` and a dependency from `Calendar` to `GoldenOrbi`. For a context with `CalendarReq`, the configuration should be [`Calendar`, `GoldenOrbi`, `CluDB`].

New types of properties. Providers can introduce new types of properties to describe the systems, such as the capacity for `EMail` services. This means a new column on the DiVA model, with specific values for each variant in the relevant rows. These new values will also affect the ranking of existing configurations.

All these changes may result in different recommendations for failure prevention and recovery. The way to keep the results consistent to these changes is to adapt the adaptation model (the DiVA model) at run-time.

In order to handle the change in the variability of the service broker, we evolve the default DiVA process as described in Figure 2(a) into an extended one as shown in Figure 2(b). This process allows to adapt the *variability at run-time*. In addition to the classical adaptation loop, a higher-level loop has been added to keep the DiVA model updated with the changes in the system as described above. In this loop, the meta-monitoring mechanism keeps a track of the higher-level changes encountered by the service broker, and feeds this information to the variability model maintainer. The latter combines the system changes and the current DiVA model, decides how to update the DiVA model, and also alters the low-level monitoring component when necessary to observe additional data. The new adapted DiVA model will be used by the DiVA reasoner as a new adaptation model for the subsequent adaptation analysis and planning.

C. Challenges

A major challenge towards adaptive variability models at run-time is to properly maintain the variability models according to the system changes. This is a typical model-system synchronisation problem in models@run-time. However, as the variability model is in a higher abstraction level than traditional run-time models, such as the context-configuration model, it is more challenging to synchronise the model with the system. We summarise the challenges as following. 1) *System heterogeneity.* There is not a standard reflection interface to obtain the run-time information from target systems. Therefore, the model maintainer need to know how to use the particular system interface. 2) *Concept mismatch.* The Variability is an abstract model, and the concepts in the model are not simply one-to-one mapped to the system concepts, as we have seen in Section III-A. 3) *Lack of knowledge.* The maintainer needs extra knowledge to understand particular phenomena observed from the running system. For example,

how to tell whether a service is impending to fail based on the performance observations.

The system-specific reflection interface, the concept mapping, and the additional knowledge form the key reference for the maintainer of the variability model to the system. Since these references are different from system to system, and may even vary in the same system when technology and user preference evolve, we need to provide a general way for developers to specify such references and to customise them when necessary.

IV. Maintaining the reference model at run-time

This section presents how we realise the adaptive variability model at run-time by extending the DiVA approach. The DiVA maintainer exploits the reflection mechanisms developed within the Broker@Cloud platform in order to obtain the run-time information needed to synchronise the DiVA model with the running system. The synchronisation behaviour of the maintainer is specified in a metamodel-based way.

A. The reflection mechanism of Broker@Cloud

A service broker provides reflection mechanism for external players to observe its run-time states. Such reflection mechanisms have two interaction styles: a *push* mechanism actively notifies observers about system changes, and a *pull* mechanism provides a passive interface for observers to retrieve the information they are interested in.

Broker@Cloud push-styled event-based reflection mechanism follows a publish-subscribe (pub-sub) pattern. An event indicates that a change has happened either in the broker platform (*e.g.*, a new service is registered), or in the individual services (such as regular events for the CPU occupation from a particular service). The broker contains a central pub-sub server, with a predefined topic to publish platform-level events. In addition, any service provider can create a new topic for its own service, through which it can publish events generated by the service itself, or by a third-party monitoring mechanism.

The pull-styled reflection of Broker@Cloud is based on linked open data and SPARQL queries. One of the innovations in the Broker@Cloud project is the usage of linked open data for service specification. Below is a specification snippet for the GoldenOrbi Service.

```
sp:GoldenOrbi a usdl-core:Service;
  usdl-core-cb:hasServiceModel sp:OrbiModel
  usdl-core-cb:dependsOn sp:CluDB
  usdl-core-cb:dependsOn sp:EMail
  sp:hasResponseTime [gr:hasValue "200"];
sp:hasResponseTime
  rdfs:subPropertyOf
    gr:quantitativeProductOrServiceProperty;
  rdfs:domain sp:ServiceModel;
  rdfs:range sp:AllowedResponseTime;
sp:AllowedReponseTime rdfs:subClassOf
  gr:QuantitativeValueInteger;
  gr:hasUnitOfMeasurement "C26"^^xsd:string;
  gr:hasMinValue "0"^^xsd:int;
  gr:hasMaxValue "1000"^^xsd:int;
  usdl-core-cb:higherIsBetter "false"
```

It specifies that *GoldenOrbi* is a service, belongs to the service model *OrbiModel*, depends on two other services, and has a response time of 200 ms. The triples later defines the property *hasResponseTime* and its type, *AllowedResponseTime*. We will not go into details of linked open data, but only emphasise the following two characteristics. First, the specification is self-contained. Any concepts used in the specification is defined in a reachable repository. Second, the specification is open and flexible. For example, a service provider is free to define a new property (including new namespaces) another *sp:hasCapacity*, and allocate a value, the corresponding value for *GoldenOrbi*. These two characteristics enable the run-time evolution of the system's variability.

B. Mapping running systems to DiVA models

A mismatch might occur when the data obtained from the reflection mechanism do not directly correspond to the property to update in the reference model/DiVA model. For example, in the service specification, the response time is defined as a concrete integer (*e.g.*, 200), while in a DiVA model it is typically at an abstract level (*e.g.*, High). Another example, the run-time measurement such as CPU occupation, memory consumption, etc., do not directly indicate failure likelihood. We employ two techniques to handle this mismatch problem: (i) Complex Event Processing (CEP) for the event-based reflection and (ii) SPARQL queries for the pull-based reflection.

Complex Event Processing aggregates low-level events into higher level ones. If some recent events combined conform to a particular pattern, the engine will create a new event using the information from the detected ones. A CEP rule defines an event pattern and how to create the new event, and is used as the reference for the CEP engine.

```
cep {srvreg(?srvname), sysvalid()|window=10}
  =>newsrv($srvname)
cep {cpuload(?srvname,?value) groupby $srvname
  if avg($value)>0.95 | window=30}
  => impfailure($srvname, $level, "CpuOverload")
```

The code snippets above shows two sample CEP rules. The event pattern and the generation logic are defined before and after the arrow "=>", respectively. The first rule is used to monitor new services. Broker@Cloud will first emit an event named `srvreg` when a provider registers a new service, and the parameter is the service name. After that, Broker@Cloud will do a global validation checking on the service policies followed by a `sysvalid` event if the policies are consistent with each other. When the two events are observed in succession within a window of 10 seconds, the engine creates a new event named `newsrv`. In the rule definition, a "?" before a variable means extracting the value in the position (in this example, it is the first parameter of the event, *i.e.*, the service name), and store the value into the variable, and later on a "$" before a variable name means getting the current value of the variable. The second rule exemplifies how to detect the impending failures. The basic idea is the following: when the average CPU occupation of a service's host during the last 30

seconds is above 95%, we create an `impfailure` event for the service, and the reason is "CpuOverload".

We use SPARQL to extract information from the service specifications, and also to do aggregation and calculation on the extracted data, in order to use them for updating the DiVA model. The following two examples illustrate this usage.

```
query qdim(?srvname): SPARQL#
  "SELECT ?modname WHERE{
   sp:$srvname usdl-core-cb:hasServiceModel
     sp:?modname"
  =>qdimo($srvname, $modname)
query qpval(?srvname, ?propname): SPARQL#
  "SELECT (?value-?min)*5/(?max-?min)
     AS ?level WHERE{
    sp:$srvname sp:$propname ?value .
    sp:$propname rdfs:range ?type .
    ?type gr:hasMinValue ?min .
    ?type gr:hasMaxValue ?max .
  }
=> qpvalo($srvname, $propname, $level)
```

The first example shows how to get the service model defined for a service, which is related to the dimension in DiVA. We name the query `qdim`, with a parameter `srvname` for the service name. The SPARQL query after that finds the triple that defines the service model for the required service, and the last item of this triple is the service model name. The output of this query is a new event named `qdimo`, with parameters of the service and its model's names. The second example shows how to retrieve a property value for a specific service (such as `GoldenOrbi`'s `responseTime`). The query first finds the triple with the requested service name, the property and the value. After that, it looks for the type of this property, and the max and min value defined in this type. The max and min values are used to normalise the concrete value into an abstract level within 0 and 5, preparing it to be applied as an abstracted DiVA property value.

C. Metamodel-driven synchronization

The main technical part to realise adaptive variability models at run-time is to automatically synchronise the variability model to the running system. Following our previous work towards an automatic run-time model construction framework [10], we implement such causal connection maintenance in a metamodel-driven way. The basic idea is to add annotations to the metamodel of the run-time model. These annotations specify how the model changes of different kinds of elements, attributes, and references are related to the system changes obtained via events or queries. The annotated metamodel is used as the reference to drive an automatic engine that maintains the causal connection.

Listing 1 shows an excerpt of the causal connection specification between the DiVA metamodel and Broker@Cloud. The DiVA metamodel is directly reused from our previous approaches, and the excerpt covers three classes in the metamodel, *i.e.*, `Variant`, `Dimension` and `DiVAModel`. The meaning of these classes can be found in Section III-A.

Before diving into the detail of each sample specification, we first briefly present the syntax of the specification language. We use the following concrete syntax to define the metamodel: A class is defined with a keyword `class` and an identifier for the class name; An attribute is defined inside its hosting class by an attribute name followed by a colon and the attribute type; A reference is defined in the same way, but starting with a keyword `ref`. Annotations are defined inside the bracket of the related class, attribute or reference. We will not exhaust the different types of annotations, but only introduce the ones appeared in the following examples. An annotation may use an event or a query, such as the ones we introduced in the previous section. Finally, the specification uses a set of variables to store intermediate data. A variable is used together with a symbol to indicate the direction of the data flow: "?" and "$" signify to store and use the variable value respectively, as we described for the CEP rules, and "!" implies to match the variable value with the context value.

Listing 1. Metamodel-driven specification of causal connection

```
1  class Variant{
2    @create on newsrv(?srvname)
3    @query qdim($srvname)
4    name: String {@id, @value=srvname}
5    for $p in owner.owner.properties:
6      @query qpval(name, $p.name)
7    @query qpval(name, $pname)
8      on newprop(?pname)
9    ref propvalue: PropValue* {
10     on qpvalo(!name, ?propname, ?v)
11     @value += PropValue{
12        name = $propname,
13        value = $v }
14     }
15     on impfailure(!name, ?value)
16     @value += PropValue{
17       prop="failurelikelihood", value=?value}
18     }
19 }
20 class Dimension{
21   @create on qdim(?srvname, ?modname)
22   name: String{@id, @value=$modname}
23   ref variant:Variant{
24     @value += Variant{name=$srvname}}
25 }
26 class DivaModel{
27   @query allservices()
28   variables : ContextVariable*{
29     on newsrv(?srvname)
30     @value+=ConextVariable{
31       name="$srvnameAvail"}
32   ref properties : Properties*
33 }
```

We first introduce the annotations in class `Variant`. The first annotation `@create` (Line 2) indicates that a variant element may be created if a `newsrv` event is captured. The service name will be extracted from the event and stored into the variable `srvname`. As a consequence of creating an variant element, a query `qdim` (as described in the previous section) will be requested to check the dimension of this variant. The result of the query will not directly be used here, but in the definition of class `Dimension` (at Line 21).

Variant has an attribute name, which is annotated as the identifier (@id). The value of this attribute is assigned by the service name extracted from the newsrv event and stored in the variable srvname. After that, we define a multi-valued reference named propvalue, which records the property values for a variant instance, such as price and storage as shown in Table II. We use two @value annotations to define two different ways of inserting PropValue elements into this reference. Firstly, at Line 10, for each of the properties defined in the DiVAModel (*i.e.*, the columns like price, storage in Table II), we launch a query qvalue (as defined in the previous section), and immediately get the response of this query, with the queried result stored in variable v. From each response, we create a new PropValue element using the property name and the queried value. If there is already a PropValue element with the same name, we will simply update its value. Secondly (Line 16), if the engine captures an impfailure event, it will create a new PropValue element named "failureLikelihood" with the value from the event.

The definition of Dimension and DiVAModel are similar. A Dimension element is created when the engine captured an event qdim (Line 21), which is the result of the query launched in Line 3. The name of the element is the service model name extracted from the qdim event, and an variant instance whose name equals to the server name in the event will be added to the variant reference. For DiVAModel, we only show the affectation of context variable. A variable is created and put under the variable reference of the DiVA model when a new service is published (Line 30). Also, we add a @root annotation to DiVAModel, as a result, the engine will create a DiVAModel element after it is launched. When the root element is created for the first time, we launch a allservices query, which triggers a set of newsrv events to initialise the variables.

At run-time, the causal connection specification will be used as a reference by the general model-system synchronisation engine. As a whole, the engine coordinates all the annotations based on the following principles. 1) Once launched, the engine initiates an element in the root type. 2) When any event is captured, it looks for the classes with a @create annotation matching this event, and create an element for each of these classes. If the new element has the same name than an existing one it simply merges the two. 3) For any newly created element, it finds the @query annotations and launch the query. 4) For any captured event, it looks through all the model elements whose type has an annotation matching the event. For each of these elements, it updates their corresponding attributes or references based on the event.

D. Examples of DiVA model maintenance

In this section we describe how the DiVA model is updated when the four changes presented in Section III-B occur, and how they affect failure prevention and recovery.

Changing property value. When a service (*e.g.*, GoldenOrbi) is overloaded, its abnormal performance will be detected by the CEP engine, which will compose a impfailure event. The event is captured by Line 15 of Listing 1, and failurelikelihood of the GoldenOrbi variant will be updated to high. This will force the DiVA reasoner to lower down the ranking of GoldenOrbi when relevant.

New alternative service. Once a service provider has registered a new FreeEmail service, and the broker has passed the self-validation, a newsrv event will be composed by the CEP engine. This event will be captured by Line 2 and Line 26, and will result in the creation of new variant and context variable, respectively. After the new FreeEmail variant is created, the engine queries all the property values of this service (Line 5) and uses the resulted values to create propvalue elements (Lines 10-14). The engine will also query the dependencies of this new service, and compose the dependency expressions (which is not exemplified in Listing 1). The reasoner will include this new FreeEMail variant immediately when computing recommendations.

A new category of services to place a new service. If a Calendar service is registered under the new service model CalDim, the corresponding newsrv event will trigger a query (Line 3), and the result goes to Line 21, and a new dimension will be created.

New types of properties. A new property will be represented by a newpro event, which triggers a new query qpval (Lines 7-8), and finally will result in a new PropValue element created on Line 11, for each Variant element. If it is a provider-specific property, it will not be applicable to most existing services and the value will be set to 0. The variable will be initialized with a neutral value 1 for the priority rules.

E. Post-synchronisation actions

Once the DiVA model is updated, the DiVA maintainer will compute the actual changes and notify the the DiVA reasoner and the monitoring component.

The DiVA reasoner will then re-compute the recommendations. In theory, the reasoner should react to any change, analyse whether the change impacts the current recommendation, and redo the constraint solving accordingly. However, in our approach, we employ a pragmatic strategy. The reasoner always re-computes on the structural changes (such as new Variants or Dimensions added), and the property value changes on failure likelihood. Other property changes will not immediately cause the re-computing of recommendations, but later when other changes happen or when the consumer requires a new recommendation.

The monitor component reacts to the addition and removal of context variables, and subscribes or un-subscribes to the corresponding topics from the pub-sub system, respectively. The monitor provides a built-in API for such operations.

V. Implementation and Evaluation

The presented approach is implemented and validated by the application in the Broker@Cloud failure recovery and prevention tool. The tool is mainly implemented as a set of Eclipse

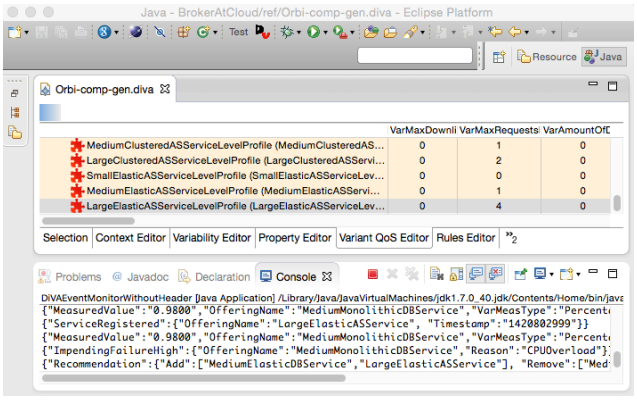

Figure 3. Snapshot of failure prevention and recovery mechanism.

plugins, extended from the original DiVA implementation. The communication between it and the other part of Broker@Cloud is implemented by RESTful APIs, and by event subscription based on the WSO2 Enterprise Service Bus[1]. The complex service processing part is implemented based on WSO2 CEP[2]. The SPARQL query is implemented on Apache Jena. The entire implementation is opensource and available on Github, as part of the Broker@Cloud solutions[3].

Figure 3 shows a sample snapshot of the tool's GUI to illustrate how the failure prevention and recovery mechanism works. The upper part presents the current generated and updated DiVA model, via a set of different views, and the currently opened view shows the quality values of variants. In the lower part, we employ a simplified event monitor to list the events received or produced by the DiVA maintainer. The event monitor console is specially customised to filter out detailed events (such as most of the intermediate events, as well as the events for query results), and present the events in a concise way. The latest events illustrates a scenario as follows. The tool first received a set of monitoring events on average CPU occupation during the time between two events, with a high reading value. These event increased the average CPU occupation in a longer period. After that the maintainer received an event about a newly registered service, followed by a validation result. A new variant was created after these two events, which is shown as the last line in the upper editor. After that, new events arrived with high CPU occupation, which finally made the average value exceed the threshold, and a high impending failure event is composed, with a recommendation calculated. It is worth noting that this new recommendation already included the newly registered service. Detailed scenarios and usages of the implemented tool can be found in our project deliverable [11], [12].

The approach has been integrated into two cloud service brokerage platforms, the Orbi Broker and the CAS Open. A set of user studies has been conducted, which proves that the Failure Prevention and Recovery mechanism is able to suggest the proper alternative solutions when failures occur or impend to occur. Details of these use cases can be found in our evaluation deliverable [13].

In order to test the performance of the mechanism inside the final platform, a set of service specifications was generated based on the sample services from CAS Open. 10 sets of test data were generated, with the numbers of services vary from 32 to 624. On each data set, we generate 5 different failures, and after each failure, we launch the failure prevention and recovery mechanism to provide new recommendations, and record the latency. The average latencies for these 10 data sets kept between 1.5 and 3.0 second, and the increase of latency according the size of data is linear. Details of the performance testing can be find in [13]. The performance is reasonable. In the worst case, 3-second latency for 600 services is still within the acceptable level.

VI. Related Work

Multi-layered architecture have been applied to the design of self-adaptive system for many years. In particular, a specific form composed of three layers has been widely adopted in several projects. In the approach proposed by Sykes et al. [14], the first layer, at the lower level of abstraction, consists of component assemblies. This layer can be adapted by the second layer which is composed of a set of adaptation plans. Finally, the third layer can tune the set of plans deployed in the underlying layer according to a set of goals. This architecture has also been applied in the cloud domain. In [15], the second layer, a learning component, continuously updates the knowledge base of the first layer, which is a fuzzy logic controller. The architecture implemented in [16] also relies on three layers but exploits models at run-time. Level 1 is composed of mechanisms to adapt the deployment of cloud-based systems based on the current workload and context of the running system, level 2 is responsible for adapting level 1 based on its long-term vision of the workload evolution. In these approaches, from the second layer above, human administrators are involved to manipulate the artefacts. By applying models@run-time approaches, we realise an automated second-layer, where the run-time model, and therefore the adaptation behaviour, is automatically updated according to the system changes.

The concept of models@run-time has been implemented by many approaches, on different types of systems, as summarized by the roadmap [3] and the survey paper [17]. Due to the heterogeneity between target systems, models@run-time are usually implemented in an ad hoc manner, based on a particular reflection mechanism provided by the target system. This paper shows a metamodel-driven way to achieve a generic specification of relations between the run-time model and the target system's reflection APIs, and is able to unify different types of mechanisms, such as events and queries. Another contribution of this work to the model@run-time community is the theory and case for multi-layered models@run-time. As a support of this contribution, we also propose a novel models@run-time pattern, based on extending the concept

[1] http://wso2.com/products/enterprise-service-bus/
[2] http://wso2.com/products/complex-event-processor/
[3] https://github.com/SingularLogic/BrokerAtCloud

of transformations. The EUREMA [18] framework supports the design and adaptation of self-adaptive systems that may involve multiple feedback loops. Developers explicitly model these feedback loops, their runtime models, their usage, the flow of models operations as well as the relationships between these models. These models are kept alive at runtime and can be evolved. This approach does not offer explicit support for controlling the monitoring and enactment transformations. These transformations could however be modelled as a feedback loops thus making our work complementary.

There are many approaches applying variability model for run-time adaptation. Our earlier DiVA approach [6] uses variability models to simulate and guide the dynamic adaptation of system configurations. Cetina et al. [19] used feature model to record the variabilities, with a definition of the impacts of features on the running system, and in this way to realise the context-based adaptation. In Parra's proposal [20], a feature model is also used to specify variability, and a composition model is proposed to define the run-time weaving of the features. In this way, they enable the run-time adaptation to be performed in the same way as adapting the system design. This paper follows the DiVA approach, and apply variability model in a new domain of run-time adaptation. However, the main contribution of this approach is to support the automatic update of variability model according to the system changes. Such a second-order adaptation is not supported by DiVA and other approaches mentioned above.

VII. Conclusion

This paper presents an approach to two-layered adaptation on cloud broker services. By maintaining causal connection between the target system and the reference model of adaptation reasoner, a two-layered models@run-time realises the automatic adjustment of the adaptation behaviour, according to system changes. The approach is motivated by, and applied to a case study of using variability models at run-time to support failure prevention and recovery on cloud service brokerage platforms. The approach is a special case of a general multi-layered models@run-time framework.

As a case study, we only provided very basic support of failure prevention and recovery. A future work is to support more sophisticated adaptation behaviours to handle failures, and investigate how the behaviour is affected by the system changes. Although we used semantic data when maintaining the variability model, the entire mismatch-handling logic is explicitly defined as CEP rules or queries. In the next step, we will introduce learning mechanisms to reduce the manual effort. Finally, we will consider generalizing and applying our approach to other domains and in particular to the Internet-of-Things (IoT). Indeed, IoT systems typically operates in the midst of the unpredictable physical world. As a result, it is impossible to anticipate all the adaptations a system may face when operating in an open context, there is a need for mechanisms that will automatically maintain the adaptation rules of a IoT system.

Acknowledgements: The research leading to these results has received funding from the European Commission's H2020 Programme under grant agreement numbers 780351 (ENACT).

References

[1] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS **4**(2) (2009). doi:10.1145/1516533.1516538

[2] de Lemos, R., Giese, H., Müller, H.A., *et al.*: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II - International Seminar, vol. 7475, pp. 1–32 (2010). doi:10.1007/978-3-642-35813-5_1

[3] Blair, G., Bencomo, N., France, R.: Models@run.time. IEEE Computer **42**(10), 22–27 (2009). doi:10.1109/MC.2009.326

[4] Fleurey, F., Morin, B., Solberg, A.: A model-driven approach to develop adaptive firmwares. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS 2011: ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 168–177 (2011). doi:10.1145/1988008.1988031

[5] Morin, B., Barais, O., Nain, G., Jézéquel, J.-M.: Taming Dynamically Adaptive Systems using models and aspects. In: ICSE 2009: 31st International Conference on Software Engineering, pp. 122–132 (2009). doi:10.1109/ICSE.2009.5070514

[6] Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., Solberg, A.: Models@Run.time to Support Dynamic Adaptation. IEEE Computer **42**(10), 44–51 (2009). doi:10.1109/MC.2009.327

[7] Floch, J., Hallsteinsen, S.O., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software **23**(2), 62–70 (2006). doi:10.1109/MS.2006.61

[8] Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-directed Architectural Change for Autonomous Systems. In: SAVCBS 2007: Conference on Specification and Verification of Component-based Systems, pp. 15–21 (2007). doi:10.1145/1292316.1292318

[9] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). doi:10.1145/505145.505149

[10] Sm@rt: Supporting Models@Runtime (2015). https://github.com/songhui/smatrt

[11] Rossini, A., Morin, B., Song, H., Veloudis, S., Dautov, R.: D40.2 – Cloud Service Failure Prevention and Recovery Components of Brokerage Framework. Broker@cloud project deliverable (July 2014)

[12] Rossini, A., Song, H., Morin, B., Høgenes, J., Veloudis, S., Paraskakis, I., Petsos, C., Simons, A.J.H., Lefticaru, R., Verginadis, Y., Patiniotakis, I.: D40.4 – Brokerage Framework: Quality Assurance and Optimisation. Broker@cloud project deliverable (October 2015)

[13] Simons, A.J.H., Armstrong, A.: D60.3: Case Study Implementation and Evaluation. Broker@cloud project deliverable (December 2015)

[14] Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: Cheng, B.H.C., de Lemos, R., Garlan, D., Giese, H., Litoiu, M., Magee, J., Müller, H.A., Taylor, R.N. (eds.) SEAMS 2008: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 1–8 (2008). doi:10.1145/1370018.1370020

[15] Jamshidi, P., Sharifloo, A.M., Pahl, C., Metzger, A., Estrada, G.: Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In: Cloud and Autonomic Computing (ICCAC), 2015 International Conference On, pp. 208–211 (2015). IEEE

[16] Ferry, N., Brataas, G., Rossini, A., Chauvel, F., Solberg, A.: Towards bridging the gap between scalability and elasticity. In: CLOSER, pp. 746–751 (2014)

[17] Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Software & Systems Modeling, 1–39 (2013). doi:10.1007/s10270-013-0394-9

[18] Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with eurema. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **8**(4), 18 (2014)

[19] Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. IEEE Computer **42**(10), 37–43 (2009). doi:10.1109/MC.2009.309

[20] Parra, C.: Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations. PhD thesis, Université des Sciences et Technologie de Lille – Lille I (2011)